THEO : an interactive proof development system

Joelle Despeyroux

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 887

THEO : AN INTERACTIVE PROOF DEVELOPMENT SYSTEM

Joëlle DESPEYROUX

AOUT 1988

RR-0887

Theo: An interactive proof development system.

Theo: Un système de développement de preuves interactif.

Joëlle Despeyroux

INRIA, Sophia-Antipolis, 2004 Route des Lucioles,
F-06565 Valbonne Cedex, France
e-mail: joelle@mirsa.inria.fr

Abstract

This paper presents the first version of a Typol interactive, tactic-driven, theorem prover called *Theo*. *Theorem prover* means here *proof développement system*. Typol is the programming language that implements Natural Semantics -a semantics pioneered by G. Plotkin under the name Structural Operational Semantics - in the Centaur meta-compiler. Centaur provides a pleasant graphic man-machine interface that Theo can use, for the user's advantage. Both the meta and the object levels of our theorem prover are logics presented in Typol. In other words, Theo is written in Typol and proves theorems of an object logic also written in Typol. Other important features of Theo are the form chosen for representing proofs, and the way proofs are performed. The internal form of the proofs is a very compact form, expressed with combinators, that is closely related to the lambda-calculus used in Automath and its descendants. Meanwhile, Theo performs proofs by a pure calculus on proofs, using a resolution rule. Proofs may be incomplete, and may contain variables. The paper presents an implementation of the Calculus of Constructions of Thierry Coquand and Gérard Huet, as an example of an object logic for Theo. It also presents the -very few- tactics that have been implemented so far. Our representation of proofs is discussed and compared with other existing forms. Differences with other existing systems are discussed at length.

Résumé

Ce papier présente la première version d'un démontrateur de théorèmes interactif appellé *Théo*. Par *démontrateur de théorèmes* nous entendons ici *système de développement de preuves*. Théo est ecrit en Typol et permet de travailler dans des logiques objets également écrites en Typol. Typol est le langage de programmation implémentant la Sémantique Naturelle -une variante de la Sémantique Operationnelle Structurelle développée par Gordon Plotkin- dans le méta-compilateur Centaur. Centaur fournit une agréable interface homme machine graphique, que Théo peut utiliser, a l'avantage de l'utilisateur.

La forme interne des preuves en Théo est une forme très compacte. Elle utilise des combinateurs, et est très proche des lambda-calculs utilisés dans Automath et ses descendants. Théo, pourtant, construit les preuves en utilisant une règle de résolution. Les preuves peuvent etre incomplètes, et contenir des variables. Le papier présente une implementation du Calcul des Constructions de Thierry Coquand et Gérard Huet, en exemple d'une logique object pour Théo. Nous présentons également les premières tactiques implémentées, et pour terminer, les différences avec d'autres systèmes.

Theo: An interactive proof development system.

Joëlle Despeyroux

INRIA, Sophia-Antipolis, 2004 Route des Lucioles,
F-06565 Valbonne Cedex, France
e-mail: joelle@mirsa.inria.fr

This paper presents the first version of a Typol interactive, tactic-driven, theorem prover called *Theo*. *Theorem prover* means here *proof development system*. Typol is the programming language that implements Natural Semantics -a semantics pioneered by G. Plotkin under the name Structural Operational Semantics - in the Centaur meta-compiler. Centaur provides a pleasant graphic man-machine interface that Theo can use, for the user's advantage. Both the meta and the object levels of our theorem prover are logics presented in Typol. In other words, Theo is written in Typol and proves theorems of an object logic also written in Typol. Other important features of Theo are the form chosen for representing proofs, and the way proofs are performed. The internal form of the proofs is a very compact form, expressed with combinators, that is closely related to the λ-calculus used in Automath and its descendants. Meanwhile, Theo performs proofs by a pure calculus on proofs, using a resolution rule. Proofs may be incomplete, and may contain variables. The paper presents an implementation of the Calculus of Constructions of Thierry Coquand and Gérard Huet, as an example of an object logic for Theo. It also presents the -very few- tactics that have been implemented so far. Our representation of proofs is discussed and compared with other existing forms. Differences with other existing systems are discussed at length.

1. Introduction

The Centaur system is a generator of programming environments based on formal semantic specifications. When given a formal specification of a language, including syntax and semantics, Centaur produces a language specific environment, including a graphic man-machine interface. From syntax specifications, one derives a parser and a syntax directed editor. Pretty-printers are also specified with a high-level formalism. Semantic specifications -static or dynamic semantics- are specified in Typol and used to derive type-checkers, translators, and interpreters/debuggers. Typol [7] is a computer language implementing Natural Semantics [6][13], a semantics pioneered by G. Plotkin[20]. A semantic specification is a collection of axioms and inference rules -a logic- and all questions about program behavior or semantic properties of a language are understood as theorems or derived rules of inference to prove in the context of this logic.

In principle Typol should be able to define any logic expressed by a Gentzen-type sequent system[9][3]. But the examples that we have treated so far, regarding semantics of sequential languages only, are all in the style of Natural Deduction[21][22], using Gentzen's sequents. The models theoretic semantics of Typol is not our concern. If you are interested in the purely deductive part of this logic, you will be interested in theorems, or derived rules. But what about the way of proving theorems? If you choose to compile your Typol program (into Prolog in the current implementation), you get the strategy of evaluation of Prolog. We clearly want to provide other strategies -tactics. In fact, the need of a *tactic-driven theorem prover* has arisen in [6], where we prove the correctness of a translation, i.e. in our framework, a derived rule, by induction on the length of the proof. At that time we were already building Typol proof-tree, but 'by hand', saying that this construction should be easily mechanized.

On an other hand, we believe that Centaur could considerably help to design a pleasant and friendly user interface for a theorem prover. As we have said, in Centaur, we can manipulate a formula as a tree, and describe logics by means of inference rules written in Typol. The theorem prover will inherit the man-machine graphic interface of the system. We will fit (almost!) 'for free' such a requirement as showing the inference rule currently involved, or designating a point in a proof-tree with the mouse, or zooming on a sub-proof-tree. Of course, there will be some features to add to the Centaur interface to adequately handle mathematical proofs in a convivial way. In fact, we have implemented the Calculus of Constructions and the Edinburgh Logical Framework in Centaur, partly to experiment the present Centaur interface.

The rest of the paper is made of six parts. We present first the very nice, powerful, and well-known logic that we chose as an example of an object logic for Theo: The Calculus of Constructions of Thierry Coquand and Gerard Huet. The second part presents the main features of our interactive Typol theorem prover. Tactics are presented in the third part. The fourth part discusses the way that we represent a proof. Finally, the last part discusses the connections with related frameworks.

2. An example: the Calculus of Constructions

Let us describe the Calculus of Constructions as an example of an object logic for Theo. The Calculus of Constructions -CC- of Thierry Coquand and Gérard Huet is a higher-order formalism for writing constructive proofs. It is aimed at formalizing mathematical reasoning and more specially type systems for programming languages. A more detailed description of the Calculus can be found in [4][12]. We will just give here the minimum knowledge necessary to understand the rest of the paper. [1]

2.1. *The language*

Terms of the Calculus are recursively defined by the following: [2]

- type i, where i is a positive integer, and $*$ are terms. [3] *kind*
- a positive integer k is a term. *bound variable*
- if m and n are terms then $m\,n$ is a term. *application*
- if m and n are terms then $\lambda m.n$ is a term. *abstraction*
- if m and n are terms then $\Pi m.n$ is a term. *product*

The definition given above defines the *deep* abstract syntax of a term, where identifiers have been translated to their de Bruijn's index. An *external* abstract syntax would have defined the abstraction, for example by the term $\lambda x : m.n$.

2.2. *The Type Inference Rules*

The rules given in Fig. 1 define the judgement $\rho \vdash m : n$, to be read as 'the term m is of type n in the environment ρ'. An environment is a list of types. By construction, the $n^{th.}$ element of an environment is the type of the $n^{th.}$ surrounding variable of the term involved. Quantifiers are all universal quantifiers -and are omitted- in a Typol rule. Finally, the rules of Fig.1 are to be read with the following declarations: m_1, n_1, p_1 : exp; a : kind; ρ : env.

---

[1] We have implemented both this Calculus and the Edinburgh Logical Framework -ELF- in Centaur. As CC and ELF have much in common -to some extent ELF is a weak predicative version of CC-, we have to choose some common notations for our implementations. The notations we have chosen are more in the flavor of the Edinburgh Logical Framework.

[2] We will forget here about constants and free variables to give a shorter description of the Calculus.

[3] We choose the preliminary terminology $*$ instead of prop (proposition), to shorten our proof trees.

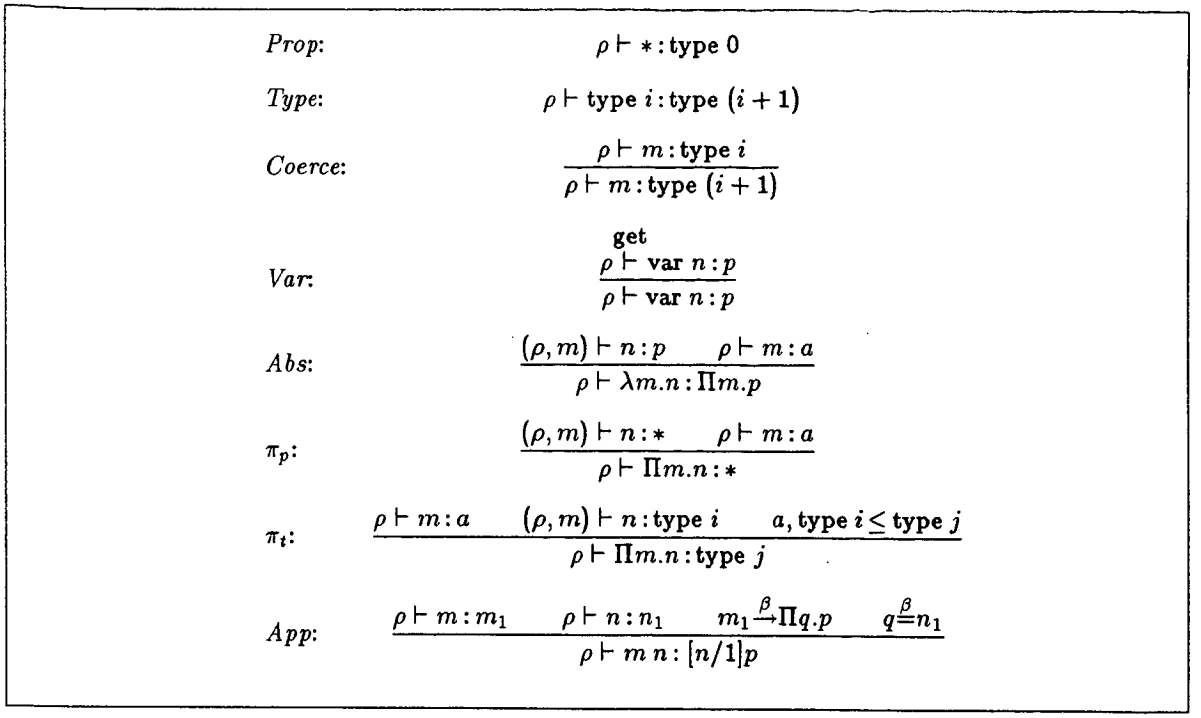

Figure 1. Type Inference Rules of CC

The definition of $\leq$, used in the 'product' rule is as follows:

$$*, \text{type } i \leq \text{type } i$$

$$\text{type } i, \text{type } j \leq \text{type}(\max(i, j))$$

Note: The inference rules of Fig.1 are the pretty-print version of Typol inference rules, except for two points. First, the variable a is declared of type kind. This means that each inference rule involving a implicitly contains the verification that a is a kind. As the dynamic type-checking is not yet implemented in Typol, the proposition kind(a) is explicitly written as a premise in the Typol program, in each inference rule involving a. The premise kind(a) will be explicit in the proof trees given in the next section. Second, functions are not yet implemented in Typol, so the notation $[n/1]p$, for example, cannot be written in a Typol program. Each function has to be specified by a set of inference rules.

3. Theo: An interactive, tactic-driven, Typol theorem prover

We present here the first Version of Theo. The tactics implemented so far are *user* and *breadth first* -a complete Prolog strategy, which is rarely implemented, because of its high costs. Theo can prove theorems but not yet derived rules, but this possibility seems easy too add. You can *undo* a proof, at any step, in Theo. Of course, you can undo as many steps as you like. This is a very pleasant tool. An even more pleasant possibility, however, would be to undo a proof, at any point in the proof. We have not yet worked on that.

Both the meta and the object levels of Theo are logics written in Typol. More work is needed to define more precisely the theoretical bases of Theo. We were first concerned with implementation issues, such as defining a compact form well suited for the proofs, and avoid the validation function of LCF. The internal form of the proofs is a very compact form expressed with combinators. Meanwhile proofs are performed by a pure calculus on proofs, using a resolution rule, as in Isabelle [18][19]. [4] Proofs may be incomplete, and may

[4] We designed Theo just before reading reports on Isabelle. On the other hand, the λ-calculus defined in the Calculus of Constructions and in the Edinburgh Logical Framework influenced us heavily for the definition of a proof in Theo.

contain variables. Finally, as tactics are for us parts of the meta-logic, it seems natural to us to implement them by means of inference rules.

3.1. *General organization*

Theo asks for an object logic written in Typol -for example a semantics of a language, or the type inference system of the Calculus of Constructions,...- and for a theorem to prove in this logic. As we are in Centaur, Theo asks in fact for two windows on these objects. The second window will be used to display the proof tree under construction. Theo then enters in a while-loop, asking for a tactic to apply and a place in the proof tree to apply it.

3.2. *Proofs and Proof Trees*

The user of Theo always sees a *proof tree*, yet Theo does not keep this proof tree but keeps instead what we shall call a *proof*. A *proof* has been defined with the aim to be the most compact form necessary to represent the current stage of a proof, in a framework where proofs are developed backwards, using a resolution rule. The definition of a proof uses combinators, and defines a calculus that is closely related to the λ-calculus used in Automath and its descendants, as we shall see later on. [5]

A proof is recursively defined by:

- $\phi \vdash \psi$. *axiom of the object logic - atomic proved term*
- $[\phi \vdash \psi]_x$, where $\phi \vdash \psi$ is a sequent of the object logic, eventually containing free variables. The subscript x is an occurrence, used by the resolution rule [6] , when developing a proof. *term to prove*
- $(R\ P_1 \cdots P_n)$, where R is the name of an inference rule of the object logic and the list $P_1 \cdots P_n$ is a list of proofs. *application*

In general, a term as above contains some variables and denotes *an incomplete proof*. A term will be called a complete proof, when it contains no 'term to prove'.

3.3. *An example of a proof development*

Consider the Calculus of Constructions -CC- described in the previous section as an example of an object logic for Theo. Suppose we look for a proof of $\Pi * .\Pi 1.2$. To that end we will try to prove the following, where p is a variable:

$$\rho_0 \vdash p : \Pi * .\Pi 1.2$$

Theo computes the corresponding -incomplete- proof, which is almost the same term: [7]

$$[\rho_0 \vdash p : \Pi * .\Pi 1.2]_a$$

The construction of the proof goes on, asking for a tactic to apply at each step. At the first step, the system may provide the following incomplete proof:

$$\begin{array}{l}(abs \\ \quad [(\rho_0, *) \vdash n_1 : \Pi 1.2]_{a_1}, \\ \quad [\rho_0 \vdash * : t_1]_{a_2}, \\ \quad [\mathrm{kind}(t_1)]_{a_3})\end{array}$$

Theo displays the corresponding proof tree, which is obtained by *pretty-printing* the previous proof. The pretty-print evaluates the proof into another form:

$$\frac{(\rho_0, *) \vdash n_1 : \Pi 1.2 \quad \rho_0 \vdash * : t_1 \quad \mathrm{kind}(t_1)}{\rho_0 \vdash \lambda * .n_1 : \Pi * .\Pi 1.2}$$

[5] See the section discussing the form of a proof.

[6] The resolution rule is described in the next subsection.

[7] The subscript a is formally an occurrence. An implementation trick consists in implementing this occurrence by a variable. The pretty-print used here reflects that trick.

As we have said previously, the user only see the proof tree. [8] So, the user points -with the mouse- to a place in the *proof tree* where he wants to apply a tactic. A hint there is that in the place where he sees a sequent to prove is in fact a tagged sequent -meaningful in the underlying *proof*. The tag is the occurrence, which enables Theo to recover the sequent that is pointed out, for applying the resolution rule to it.

The construction of the proof goes on and may provide the following proof tree:

$$\frac{\dfrac{((\rho_0,*),1)\vdash x_1:2 \quad (\rho_0,*)\vdash 1:t_2 \quad \mathrm{kind}(t_2)}{(\rho_0,*)\vdash \lambda 1.n_2:\Pi 1.2} \quad \rho_0\vdash *:t_1 \quad \mathrm{kind}(t_1)}{\rho_0\vdash \lambda *.\lambda 1.n_2:\Pi *.\Pi 1.2}$$

Finally, a proof is found, and displayed as the following proof tree: [9]

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{xi}(0,1\to 2)}{\mathrm{add}(1,1\to 2)}}{\mathrm{get}((\rho_0,1)\vdash 1:2)} \quad \dfrac{\dfrac{\mathrm{xi}(0,*\to *)}{\mathrm{add}(1,*\to *)}}{\mathrm{get}((\rho_0,*)\vdash 1:*)}}{(\rho_0,1)\vdash 1:2 \quad (\rho_0,*)\vdash 1:*} \quad \mathrm{kind}(*)}{(\rho_0,*)\vdash \lambda 1.1:\Pi 1.2} \quad \rho_0\vdash *:type_0 \quad \mathrm{kind}(type_0)}{\rho_0\vdash \lambda *.\lambda 1.1:\Pi *.\Pi 1.2}$$

The corresponding proof is much more compact:

$$\begin{array}{l}(abs\\ \quad (abs\ (var(get_1(add_1\ \mathrm{xi}(0,1\to 2))))),\\ \qquad (var(get_1(add_1\ \mathrm{xi}(0,*\to *)))),\\ \qquad \mathrm{kind}(*)),\\ \quad \rho_0\vdash *:type_0),\\ \quad \mathrm{kind}(type_0))\end{array}$$

Of course, there is much less differences between the size of these two terms when they are pretty-printed in Tex. But our terms are Typol trees. So to compare two terms, we must compute their sizes, i.e. the number of the nodes in the trees. The gain become then much more evident. The above proof tree is one third as large than its corresponding proof. The gain is much greater for a larger proof tree. Each application of a rule in a proof costs the number of nodes of the node 'application of a rule name to'. Each application of a rule in a proof tree costs the number of nodes of the node 'inference rule with such sequent as a conclusion'. So, the gain mostly relies on the sizes of the sequents involved in a proof tree. A simple case is the one of a very large proof, for which the average size of these sequents makes sense, and for which the sizes of the terms left to prove is negligible on the number of rules already applied. In this case, the size of the proof tree is proportional to the size of its corresponding proof, with a factor of proportion of the average number of nodes of a sequent in the proof tree.

The example described so far is actually running in Centaur, under Theo. The proof may be obtained either with the 'user' or with the 'breadth first' tactic. We present these tactics in the next section. Let us first describe the primitive tool of a tactic, which says how to apply an object inference rule at some point in the proof.

3.4. *A Resolution Rule*

The Resolution Rule, which enables us to construct a proof, is the following: [10]

[8] Of course it will be easy to allow the user to see the actual proof, if he prefers so. But a large proof term may be very difficult to read, and a proof tree displays more information.

[9] Rules add and xi describe calculus on the de Bruijn's index of the terms of CC.

[10] The Typol rule is a little bit more heavy because there are not yet functions in Typol.

$$P \vdash R{:}\frac{p}{c},\ c_a \rightarrow P[(R\,p_x)/c_a]$$

Where a list L of premises subscript by a list x of occurrences stands for the list of each element of L subscript by one element of x. Remember that the subscript of a sequent in a proof denotes the occurrence of that sequent in the proof.

The above rule describes the application of an inference rule $\frac{p}{\phi}$, of name R, to a term ψ_a in a proof P. [11] P is an incomplete proof, containing some variables. First ϕ is match against ψ. Notice that this unification affects the all proof P. Then P is modified again: the sequent ψ_a is replaced by a term denoting the application of the rule R to a list of new terms to prove p_i, one for each premise of the rule. (This substitution is specified by a set of inference rules, not by β-reduction.) Except for this substitution, all the work is done by unification.

4. Tactics in Theo

We have said that for us tactics are parts of the meta-logic, and are implemented by means of inference rules. However, the Typol program that specifies them is compiled into Prolog. Thus the strategy of evaluation of prolog gives some *control* to the tactics. Gilles Kahn compares this *execution* of a logic with the transformation of a set of equational rules into *oriented* rewriting rules. This feature must be used with much care, but may be very useful. Only two tactics have been implemented in Theo, up to now: The *user* tactic, and the *breadth first* tactic. We shall describe them here.

4.1. *The user tactic*

This tactic consists in asking the user for an inference rule to apply and a place to apply it. It could be named: 'no tactic'. Notice, however, that this 'no-tactic' is the only tactic required by mathematicians for a great amount of there proofs. The user tactic is implemented by one -meta- inference rule, with two predicates that realize the desired interface with the user:

$$\frac{\overset{\text{gen}}{\vdash}\ r \rightarrow r' \quad \overset{\text{pp}}{\vdash}\ seq' \rightarrow seq \quad P \overset{\text{app}}{\vdash}\ r', seq' \rightarrow P'}{\vdash P \rightarrow P'}\ \text{ask\_rule}(r), \text{ask\_seq}(seq)$$

The side conditions of this rule are the predicates that realize the interface. ask_rule (resp. ask_seq) asks the user to point out -with the mouse- on an inference rule (resp. a sequent), and get it -that is get the corresponding abstract syntax tree. The set of rule gen 'generalizes' the inference rule, i.e. replaces Typol variables -which are identifiers- by Prolog variables. The set name 'pp' stands for pretty-print, but here it just means 'instanciate', the inverse of 'gen'. The set app contains the resolution rule previously presented.

4.2. *The breadth first tactic*

This tactic describes a complete Prolog strategy, which is rarely implemented, because of its high costs. Breadth first is defined by eighteen inference rules. It consists in building all the possible proofs for a sequent, building them with the breadth first strategy. At each step, the tactic has build a certain amont of proofs. For each proof, for each sequent to prove, for each rule that is applicable to it, the rule is applied. Each proof gives raise to a list of proofs, whenever possible. In the case where there is no applicable rule for a sequent not yet proved, the corresponding proof, which cannot be completed, is forgotten. If a -complete- proof exists, the tactic will produce it at some point. At this point, there is the choice to end with that proof, or to go on, to find other possible proofs. This is specified by two sets of rules. The first one, set choice, says that a possible result is one of the -complete- proofs found. It only contains the two following rules:

$$\vdash p.ps \rightarrow p$$

$$\frac{\vdash ps \rightarrow p'}{\vdash p.ps \rightarrow p'}$$

The second set, cont, defines the 'continuation' of the tactic, when a list of complete proofs has been produced. In the case where the list is empty, the tactic go on. In the case where some (maybe several)

proof has been completed, the tactic may either go on or stop. In the latter case the tactic choose a proof among all the possible proofs found. The set cont defines the judgment $ps \vdash ts \rightarrow proved, ts'$, which can be read as 'whenever, at some point, some complete proof(s) ps has been found, and currently built proof(s) are ts, the tactic provides the complete proof(s) $proved$, and the incomplete proof(s) ts''.

$$\frac{\vdash \quad ts \rightarrow proved, ts'}{\emptyset \vdash ts \rightarrow proved, ts'}\ \text{breadth\_1st}$$

$$p.ps \vdash ts \rightarrow p.ps, ts$$

$$\frac{\vdash \quad ts \rightarrow proved, ts'}{p.ps \vdash ts \rightarrow proved, ts'}\ \text{breadth\_1st}$$

A choice is easy to implement in Logic. As we have said in the beginning of this section, the compiler of Typol gives some control on this choice. The capability to effectively provide all the results is implemented on top of the theorem prover, using the debug facility of Typol, which enables the user to *backtrack* on its choice of a rule. So this feature of Theo is view, and implemented, as a tool of the Typol interface.

5. Discussion on the form of a proof

The reason of this section is to discuss the representation of a proof that we have chosen. To that end, we will present three other forms of a proof. The first two are slight, but interesting, modifications of the form we have chosen. The third one is the one chosen in Isabelle.

5.1. *A first form*

We give here a form of a proof, which may be prefered for certain cases. For example, let us take the first order logic, with the two following usual rules, in natural deduction:

$$\supset I: \quad \frac{\gamma, \phi \vdash \psi}{\gamma \vdash \phi \supset \psi}$$

$$\text{HypI:}\ \frac{\gamma \vdash \phi}{\gamma, \psi \vdash \phi}$$

Suppose we would like to prove: $\vdash P \supset (Q \supset P)$ in this logic. An incomplete proof can be, for example:

$$(\supset I\ (\supset I\ [P, Q \vdash P]))$$

This -incomplete- proof will be pretty-printed as:

$$\frac{\frac{P, Q \vdash P}{P \vdash Q \supset P}}{\vdash P \supset (Q \supset P)}$$

Now, we would like to apply rule HypI, and construct the following proof tree:

$$\frac{\frac{\frac{P \vdash P}{P, Q \vdash P}}{P \vdash Q \supset P}}{\vdash P \supset (Q \supset P)}$$

But we have not given a way to specify that we would like to apply the rule HypI with Q as ϕ. Here, we have three choices. The first choice is to say: well, we asked for some proof $P(Q)$, and Theo provides $P(\phi)$ instead. It is just the same proof, up to a renaming of variable. I am happy with it.

The second choice is to enable the user to do the desired substitution, if he desires so. In that case, Theo would have the ability to instanciate a variable of a proof. This feature may be interesting by itself. Therefore our personal choice is the just proposed one -not yet implemented.

Now, the user may argue that this could be done automatically by Theo. The answer is yes, but this requires some changes. Firstly, a minor change: the second inference rule will be transformed -by Theo- into the following inference rule:

$$\text{HypI}: \quad \lambda\psi.\ \frac{\gamma \vdash \phi}{\gamma, \psi \vdash \phi}$$

Then the proof could have the following form:

$$\supset I\,(\supset I\,(\text{HypI}\,(Q)\,(P \vdash P)))$$

This requires a slight modification in the definition of the application in a proof:

- $\phi \vdash \psi$. *axiom of the object logic - atomic proved term*
- $[\phi \vdash \psi]_x$, where $\phi \vdash \psi$ is a sequent of the object logic, eventually containing variables. *term to prove*
- $((R\ y_1 \cdots y_n)\ P_1 \cdots P_n)$, where R is the name of a rule of the object logic and P_i are proofs. y_i are terms to substitute for those free variables of the inference rule that appear in the conclusion without appearing in the premises. The other variables of the rule are bound through the proofs of the premises. *application*

The resolution rule must be modified accordingly:

$$P \vdash R{:}\lambda V.\frac{p}{c},\ c_a \rightarrow P[(R\,V\,p_x)/c_a]$$

The modifications are really slight modifications, but they result in increasing the size of a proof. Therefore we are not in favor of this last possibility.

5.2. *A second form*

We give here an other form of a proof, where we choose λ-terms, instead of combinators. This may be a little bit misleading as we don't do β-reduction. The interest in this form is all in the light it gives on the link between Theo and the Automath's family. This form is more in the favor of Automath and its descendants, and particularly of the Edinburgh Logical Framework.

A proof is recursively defined by: (We choose here the previous form in addition to the one presented here.)

- $\phi \vdash \psi$. *axiom of the object logic*
- x. *bound variable, denoting a non proved term*
- $\lambda(x_1 : T_1 \cdots x_n : T_n).P$, where $(x_i : T_i)_i$ is a list of bindings, with x_i being a variable and T_i being a sequent -to be proved- of the object logic , and P is a proof containing no further abstraction. *abstraction, denoting an incomplete proof*
- $((R\ y_1 \cdots y_n)\ P_1 \cdots P_n)$, where R is the name of an inference rule of the object logic and P_i are proofs containing no abstraction. y_i are terms to substitute for those free variables of the inference rule that appear in the conclusion without appearing in the premises. The other variables of the rule are bounded through the proofs of the premises. *application*

It is worth noting that the term P, P_i, or T_i in an abstraction or an application, may contains *free variables*, which have nothing to do with the *bound variables* that denote a non proved term.

Notations: The corresponding Resolution Rule requires the definitions of two functions γ and δ, whose arguments are lists of sequents of the object logic. Intuitively, $(\delta\ L)$ is the list L where each element -sequent- is prefixed with a new -free- variable. The sequent is understood as the "type" of that variable, in LF-like

logical frameworks. $(\gamma\ (\delta\ L))$ is a list of new variables, one for each sequent of the list L. $t.L$ denotes a list whose first element is t and whose rest is L. The two functions can be formally defined as follows:

- $(\delta\ T) = x : T$
- $(\delta\ t.T) = (\delta\ t).(\delta\ T)$
- $(\gamma\ x : T) = x$
- $(\gamma\ x.Y) = (\gamma\ x).(\gamma\ Y)$

The Resolution Rule is the following:

$$\lambda Y.\ \lambda x : c.\ \lambda Z.P \vdash R : \lambda V.\frac{p}{c},\ x \rightarrow \lambda Y.\ \lambda(\delta\ p).\ \lambda Z.P[x = (R\ V\ (\gamma\ (\delta\ p)))]$$

The above rule describes the application of a rule $R : \lambda V.\frac{p}{\phi}$ in x in a proof $\lambda Y.\ \lambda x : \psi_1.\ \lambda Z.P$. First ϕ is match against ψ_1. Then x, in the proof P, is match against a term denoting the application of the rule R to some terms V and some new variables $(\gamma\ (\delta\ p))$, one for each premise of the rule. Finally, the binder x is replaced by the list of bindings $(\delta\ p)$. All the work, here again, except the last part, is done by unification. With that view of proofs, when the user point -with the mouse- to a place in the *proof tree* where he wants to apply a tactic, in the place where he sees a sequent to prove is in fact a variable -meaningful in the underlying *proof*- having for type that sequent.

5.3. *A third form*

We said previously that the form of a proof has been defined with the aim to be the most compact form necessary to represent the current stage of a proof, in a framework where proofs are developed backwards, using a resolution rule. We did not know, at that time, about the form chosen in Isabelle. Isabelle represents a proof by an inference rule -which denotes a proposition in higher order logic. This form is even more compact than our. The proofs presented in this paper would have approximately the same size in Isabelle. But, for sure, this is not true for large proofs. The solution chosen in Isabelle has some advantages (compacity for example) but also some drawbacks (operations on the proof seem a-priori impossible). However, Isabelle can overcame these in some cases. For example, the extraction of programs from proofs can still be made in the Calculus of Constructions, as a program can be derived either from the proof or from the sequent proved -provided some care in the use of the rule introducing a hierarchy of types. We discuss aspects of Isabelle other than the form of a proof in the next section.

6. Related work

Related work are numerous, and most of them are quite recent systems.

Only two systems are rather old: LCF and Nuprl. LCF [17]was the first theorem prover. ML was defined as its meta-language for writing tactics. Terms, formulae and theorems are typed data. An inference rule is defined as a function from theorems to theorems. Its 'inverse' is a tactic mapping a goal to subgoals. *Tacticals* express functions on tactics, such as repetition of a given tactic. In order to obtain the proof just constructed, a *validation function* has to be evaluated; for us, this is a serious default. Theorems can only be constructed by tactics. Still, a tactic can be invalid. In this case the validation function will not return the theorem asked for. LCF has its own object theory, called PPλ, and does not -in principle- enables the user to enter its own object logic.

Nuprl [5]is still the most sophisticated system. It is based on a Constructive Type Theory, that mainly extends Martin-Löf's Intuitionistic Type Theory with recursive types and partial functions. So a proof in Nuprl is a term of a higher-order λ-calculus. Tactics are expressed by ML functions, as in LCF. Nuprl provides a user interface -less powerful than Centaur.

λ-prolog [14][16]is currently developed by the D. Miller team. It is a very interesting extension of Prolog to allow λ-terms and applications inside a term of a clause. λ-terms are evaluated by the usual β-reduction. Unification has been extended to higher order unification. λ-prolog has been recently used to implement a theorem prover [8]. Proofs are represented exactly as our proofs. But the theorem prover is not viewed as

a meta-logic manipulating an object logic. An inference rule of the object logic is defined as a tactic, more precisely a λ-prolog clause saying that the proof of the conclusion will be such provided proofs of the premises are such and such. Tactics have exactly the same shape. In a certain sense, there is only a meta-logic. We are not following this approach.

Laurent Hascoët[11], from our team, begun last year some experiments in theorem proving under Centaur. His theorem prover consists in prolog clauses that manipulates formally Typol rules. So far the differences between his system and Theo could be seen as minor, as far as design is concerned, as Theo consists in Typol rules that manipulates formally other Typol rules. Note however that writing Theo in Typol enables us to fit in with the idea that the theorem prover is nothing else but the meta-logic in which proofs are performed. An other difference is the form chosen for representing a proof. His system keeps the full proof tree -which can be very large as we have seen- whereas Theo defines its own form of a proof -as all others systems do.

The Calculus of Constructions, Isabelle, and the Edinburgh Logical Framework are three recent systems. While the Calculus of Constructions [4][12]-CC- is aimed at formalizing mathematical reasoning, the Edinburgh Logical Framework -ELF- and Isabelle are general frameworks to represent logics. While a proof in CC and ELF is a term of a higher-order λ-calculus, a proof in Isabelle is a proposition of higher-order logic, representing a derived rule of the object logic.

ELF [10]identifies sequents and inference rules in a product. So one can write $n^{th.}$ order sequents. We have not that view of sequents. But maybe that is because we never had the need of writting a third or more order sequent. ELF is aimed at allowing uniform presentations of logics. According to the designer of ELF [2], 'it is an 'ELF thesis' that well-behaved natural-deduction formalisms are those that can be directly encoded in the ELF'. The price to pay is to prove that the object logic has been correctly translated. This can mean for example that 'there is a compositional bijection between proofs in the natural deduction system and proof terms in ELF'. It is true that the same problem exists in Isabelle; but the proof seems much easier there. We also change the presentation of an object logic, but this seems to be a very slight modification: we have no side conditions in a rule, [12] as -with our view of sequents- there are nothing else than premises. A prototype version of ELF as been implemented under the Cornell's Synthetizer by T. Griffin.

Isabelle [18][19]is based on higher-order logic. A proof is a proposition of the meta logic -higher-order logic-, that represents a derived rule of the object logic. The application of a rule is defined by a Resolution Rule. This rule, as well as the 'lifting' rule can be derived in the meta logic. A rule of the object logic is represented by a term in the typed λ-calculus. Tactics are expressed by ML functions. It is due to the use of unification in the process of building proofs that no validation function is needed, despite the use of ML. Because Isabelle does not keep the full proof, it seems a-priori not possible to do some operations on the proof. If, despite that, we ever choose Isabelle proofs, our resolution rule could be the same than the one of Isabelle (the one presented p.28 in [19]). In fact the theoretical foundations of Theo could then be those of Isabelle. The only remaining difference will then be on the form of tactics, which, in Theo, are expressed in Typol.

7. Conclusion

We presented here the first version of a Typol, tactic driven, theorem prover, and discussed its choices at length, comparing it with other systems. An evident advantage of Theo is that it is build on top of Centaur, using the man-machine graphic interface of the system. Beside this, we think that Theo as two major advantages. The first one is the very compact form used to represent proofs. The second one is its design: a logic -presented as a Typol program- that manipulates formally an object logic -also given by a Typol program. These two advantages are still to be confirmed. The design of Theo seems a good design to us, but still, we have to precise the theoretical foundations of Theo.

Further developments would provide the user with a richer interface, and much more tactics, to adequately handle mathematical proofs in a convivial way. An important area of research also concerns

[12] We only use side conditions for interface purpose. In fact we sometimes -dare to- write some premises as side conditions, but they are still treated as premises, at least in the current version of Typol, and Theo.

higher-order unification and quantifiers. For the moment Typol only knows about first-order unification and universal quantifiers. Finally, some tools as proving a derived rule, and adding it in the object logic in order to use it for future proofs seems easy to implement, and should be implemented first.

Acknowledgements. Thanks go to the Formel team, and especially to Thierry Coquand, Philippe Lechenadec and Christine Paulin-Mohring who help me -sometimes by electronic mail- to understand the Calculus of Constructions.

References

[1] A. AVRON, "Simple Consequence Relations" Edinburgh Report ECS-LFCS-87-30, June 1987.

[2] A. AVRON, F. HONSELL, A. MASON, "Using typed λ-calculus to implement formal systems on a machine", Edinburgh Report ECS-LFCS-87-31, July 1987.

[3] J. BARWISE, "The Handbook of Mathematical Logic", North-Holland, Amsterdam, 1977, reprinted in 1983.

[4] TH. COQUAND, "An analysis of Girard's paradox" Proc. of the first ACM-IEEE Symp. on Logic In Computer Science, Cambridge, Ma, USA, June 1986.

[5] R. L. CONSTABLE and all, "Implementing Mathematics with the Nuprl Proof Development System", Prentice-Hall, 1986.

[6] J. DESPEYROUX, "Proof of translation in natural semantics", Inria Report 514, april 1986. also in the Proc. of the first ACM-IEEE Symp. on Logic In Computer Science, Cambridge, Ma, USA, June 1986.

[7] T. DESPEYROUX, "Typol: a formalism to implement Natural Semantics", INRIA Report 94, mars 1988.

[8] A. FELTY, D. MILLER, "Specifying theorem provers in a higher-order logic programming language", Ninth Conference on Automated Deduction, 1988, and Report MS-CIS-88-12, University of Pennsylvania, February 1988.

[9] G. GENTZEN, "The Collected Papers of Gerhard Gentzen", E. Szabo, North-Holland, Amsterdam, 1969.

[10] R. HARPER, F. HONSELL, G. PLOTKIN, "A Framework for defining Logics", Proc. of the second ACM-IEEE Symp. on Logic In Computer Science, Cornell, USA, 1987.

[11] L. HASCOËT, "A tactic-driven system for building proofs", INRIA report 770, Dec. 1987. also in the proc. of the 7th seminar "Programmation en Logique", Tregastel, May 1988.

[12] G. HUET, "A uniform approach to Type Theory", February 1988.

[13] G. KAHN, "Natural Semantics", Proc. of Symp. on Theorical Aspects of Computer Science, Passau, Germany, February 1987, also INRIA Report 601, Feb. 1987.

[14] D. MILLER, G. NADATHUR "A logic programming approach to manipulating formulas and programs", Report MS-CIS-87-113, University of Pennsylvania, December 1987.

[15] CH. MOHRING, "Algorithm development in the calculus of constructions", Proc. of the first ACM-IEEE Symp. on Logic In Computer Science, Cambridge, Ma, USA, June 1986.

[16] G NADATHUR, "A higher-order logic as the basis for logic programming", Ph.D. dissertation, University of Pennsylvania, Dec. 1986, also Report MS-CIS-87-48, University of Pennsylvania, June 1987.

[17] L.C. PAULSON, "Logic and computation. Interactive proof with Cambridge LCF", Cambridge Tracts in Theoretical Computer Science 2, 1987.

[18] L.C. PAULSON, "The representation of logics in higher-order logic", Cambridge Technical Report 113, 1987.

[19] L.C. PAULSON, "The foundation of a generic theorem prover", Cambridge Technical Report 130, march 1988.

[20] G.D. PLOTKIN, "A structural approach to operational semantics", Aarhus Report DAIMI FN-19, 1981.

[21] D. PRAWITZ, "Natural Deduction, a Proof-Theoretical Study", Almqvist & Wiksell, Stockholm, 1965.

[22] D. PRAWITZ, "Ideas and results in Proof Theory", Proc. of the 2nd. Scand. Logic Congress, North Holland, 1971.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique